# Bounded Seas

Jan Kurš*, Mircea Lungu, Rathesan Iyadurai, Oscar Nierstrasz

*Software Composition Group, University of Bern, Switzerland*
*http://scg.unibe.ch*

**Abstract**

Imprecise manipulation of source code (semi-parsing) is useful for tasks such as robust parsing, error recovery, lexical analysis, and rapid development of parsers for data extraction. An island grammar precisely defines only a subset of a language syntax (islands), while the rest of the syntax (water) is defined imprecisely.

Usually water is defined as the negation of islands. Albeit simple, such a definition of water is naïve and impedes composition of islands. When developing an island grammar, sooner or later a language engineer has to create water tailored to each individual island. Such an approach is fragile, because water can change with any change of a grammar. It is time-consuming, because water is defined manually by an engineer and not automatically. Finally, an island surrounded by water cannot be reused because water has to be defined for every grammar individually.

In this paper we propose a new technique of island parsing — bounded seas. Bounded seas are composable, robust, reusable and easy to use because island-specific water is created automatically. Our work focuses on applications of island parsing to data extraction from source code. We have integrated bounded seas into a parser combinator framework as a demonstration of their composability and reusability.

*Keywords:* semi-parsing, island parsing, parsing expression grammars

*Corresponding author
*Email address:* kurs@iam.unibe.ch (Jan Kurš)

## 1. Introduction

Island grammars [1] offer a way to parse input without complete knowledge of the target grammar. They are especially useful for extracting selected information from source files, reverse engineering and similar applications. The approach assumes that only a subset of the language syntax is known or of interest (the islands), while the rest of the syntax is undefined (the water). During parsing, any unrecognized input (water) is skipped until an island is found.

A common misconception is that water should consume everything until some island is detected. Rules for such water are easy to define, but they cause composability problems. Consider a parser where local variables are defined as islands within a method body. Now suppose a method declaring no local variables is followed by one that does. In this case the water might consume the end of the first method as well as the start of the second method until a variable declaration is found. The method variables from the second method will then be improperly assigned to the first one.

In practice, language engineers define many small islands to guide the parsing process. However it is difficult to define such islands in a robust way so that they function correctly in multiple contexts. As a consequence they are neither reusable nor composable.

To prevent our variable declaring island from skipping to another method, we have to make its water stop at most at the end of a method. In general, we have to analyze and update each particular island's water, depending on its context. Yet island-specific water is fragile, hard to define and it is not reusable. It is fragile, because it requires re-evaluation by a language engineer after any change in a grammar. It is hard to define, because it requires the engineer's time for detailed analysis of a grammar. It is not reusable, because island-specific water depends on rules following the island, thus it is tailored to the context in which the island is used — it is not general.

In this paper we propose a new technique for island parsing: *bounded seas* [**?**

]. Bounded seas are composable, reusable, robust and easy to use. The key idea of bounded seas is that specialized water is defined for each particular island (depending on the context of the island) so that an island can be embedded into any rule. To achieve such composability, water is not allowed to consume any input that would be consumed by a following rule.

To prevent fragility and to improve reusability, we compute water automatically, without user interaction. To prove feasibility, we integrate bounded seas into Petit Parser [2], a PEG–based [3] (see Appendix A) parser combinator [4] framework.

In addition to our previous work [**?** ] we evaluate the usability of bounded seas in two case studies, we present a performance study, and we provide more details about the implementation. The contributions of the paper are:

- the definition of bounded seas, a composable, reusable, robust and easy method of island parsing;
- a formalization of bounded seas for PEGs;
- an implementation of bounded seas in a PEG-based parser combinator framework; and
- case studies of semi-parsing of Java and Ruby using bounded seas.

*Structure.* Section 2 motivates this work by presenting the limitations of island grammars with an example. Section 3 presents our solution to overcoming these limitations by introducing bounded seas. Section 4 introduces a sea operator for PEGs, which creates a bounded sea from an arbitrary PEG expression. Section 5 presents our implementation of bounded seas in PetitParser. Section 6 discusses the applicability of bounded seas to GLL parsers, design decisions and some limitations of bounded seas. Section 7 analyzes how well bounded seas perform compare to other island parsers. Section 8 analyzes usability of bounded seas for context-sensitive grammars, particularly for indentation-sensitive grammars. Section 9 surveys other semi-parsing techniques and highlights similarities and

differences between them and bounded seas. Finally, section 10 concludes this paper with a summary of the contributions.

## 2. Motivating Example

Let us consider the source code in Listing 1 written in some proprietary object-oriented language. We don't have a grammar specification for the code, because the parser was written using *ad hoc* techniques, and we do not have access to its implementation. Let us suppose that our task is to extract class and method names. Classes may be contained within other classes and we need to keep track of which class each method belongs to.

```
class Shape
    Color color;

    method getColor {
        return color;
    }
    int uid = UIDGenerator.newUID;
endclass
```

Listing 1: Source code of the `Shape` class in a proprietary language.

### *2.1. Why not use Regular Expressions?*

To extract a flat list of method names, we could use regular expressions. We need, however, to keep track of the nesting of classes and methods within classes. Regular expressions are only capable of keeping track of finite state, so are formally too weak to analyze our input. To deal with nested structures, we need at least a context-free parser.

Modern implementations of regular expression frameworks can parse more than regular languages (*e.g.*, using recursive patterns[1]). Such powerful frameworks can handle our rather simple task. However regular expressions are not meant to specify complex grammars since they tend to be hard to maintain when the complexity of the grammar grows.

[1] `http://perldoc.perl.org/perlre.html`

```
start       ← class
class       ← 'class' id classBody 'endclass'
classBody   ← methodWater

methodWater ← (!'method' .)* method (!'endclass'.)*

method      ← 'method' id block
block       ← '{' (!'}' .)* '}'

id          ← letter (letter / number)*
letter      ← 'a' / 'b' / 'c' / ...
number      ← '1' / '2' / '3' / ...
```

Listing 2: Our first island grammar.

### *2.2. A Naïve Island Grammar*

To write a parser, we need a grammar. Because the grammar can easily consist of a hundred rules (*e.g.*, ≈ 80 for Python, ≈ 180 for Java) and since we are only interested in specific parts of the grammar, we define an island grammar as a PEG (see Appendix A) with fewer than ten rules as in Listing 2. We initially assume that each class body contains just one method.[2] Since we are interested in extracting method names, we define the `method` rule as an island inside of the `methodWater` rule which surrounds it with water. The `methodWater` rule is defined imprecisely: water skips everything until the string `"method"` is found.

We also define the `block` rule, which consumes an open curly brace and then skips everything until the closing curly brace is found.

The `methodWater` rule in the grammar in Listing 2 uses a naïve definition of water. It will work as long as we do not complicate the grammar.

[2]We use an almost standard PEG formalism for grammar definitions (see Appendix A). A terminal is quoted `'terminal'`, a non-terminal is not quoted `nonterminal`, a sequence is a concatenation of expressions, prioritized choice is marked as `/`, repetition as `*`, a not-predicate as `!`, and `.` stands for any character.

*2.2.1. Composability Problems.*

Suppose that in order to allow multiple classes in a single file we modify the start rule to allow repetition ( `start ← class*` ). Parsing the input in Listing 3 should fail because `Shape` does not contain a method. The result, however, no matter whether we use PEG or CFG, is only one class — `Shape` (instead of `Shape` and `Circle` ) — with a method `getDiameter` , which is wrong. We see that our water is too greedy here, trying to find a `method` at any cost and ignoring the `'endclass'` and the `Circle` definition.

```
class Shape
    int uid = UIDGenerator.newUID;
endclass

class Circle
    int diameter;

    method getDiameter {
        return diameter;
    }
endclass
```

Listing 3: Source code of `Shape` and `Circle` classes.

Things do not get better when we allow multiple repetitions of `methodWater` within `classBody` ( `classBody ← methodWater*` ). The parser will stay confused, and, depending on the technology (CFG, PEG), the result will be either ambiguous (CFG) or incorrect (PEG).

The language engineer has to use either a) disambiguation rules and filters [5, 6] to filter out unwanted results of CFGs; or b) predicates to prevent the incorrect decisions of CFGs and PEGs. Since predicates are applicable for both technologies (CFGs and PEGs), we focus on this approach.

*2.3. An Advanced Island Grammar*

To make the `methodWater` rule composable we must make it possible for it to be embedded into optional ( `?` ) or repetition ( `+` , `*` ) rules. We consequently define the grammar as in Listing 4. This new definition can properly parse multiple classes in a file with an arbitrary number of methods in a class. We

achieve composability by forbidding the water to go beyond the `'endclass'` keyword and by forbidding the water to consume any method definition.

```
start       ← class*
class       ← 'class' id classBody 'endclass'
classBody   ← (methodWater)*

methodWater ← (!'method' !'endclass'.)*
                method
              (!'method' !'endclass'.)*

method      ← 'method' id block
block       ← '{'
              (
                (!'}' !'{' .)*
                  block
                (!'}' !'{' .)*
              )*
              '}'

id          ← letter (letter / number)*
letter      ← 'a' / 'b' / 'c' ...
number      ← '1' / '2' / '3' ...
```

Listing 4: Complete and final island grammar.

One can see that the syntactic predicates in the `methodWater` are more complicated. They have been inferred from the rest of the grammar by analyzing which tokens can appear after the `method` island. In case we decide to allow for nested classes, *i.e.*, if we extend the rule `classBody` to:

```
classBody ← (methodWater / classWater)*
```

we have to revise the predicates of `methodWater` to add `!'class'`, and we have to find the proper predicates for the `classWater` rule.

*2.3.1. Ease of Use, Robustness, and Reusability Problems.*

The limitations of defining `methodWater` and `classWater` by hand illustrate the general problems of semi-parsing [7, 8] with island grammars:

1. Water rules are hard to define correctly because they require the entire grammar to be analysed.

```
class     ← 'class' id classBody 'endclass'
classBody ← methodSea*

methodSea ← ~method~
method    ← 'method' id block

block     ← '{' ~(block / ϵ)~* '}'

id        ← letter (letter / number)*
letter    ← 'a' / 'b' / 'c' ...
number    ← '1' / '2' / '3' ...
```

Listing 5: Island Grammar from Listing 4 rewritten with the sea operator.

2. The definition of water is fragile because predicates need to be re-evaluated after any change in a grammar.

3. Finally, the water rules are tailored just for a specific grammar and cannot be reused in another grammar with different rules.

## 3. Bounded Seas

### *3.1. The Sea Operator in a Nutshell*

We have shown that water must be tailored both to the island within the water and to the surroundings of the water (*e.g.*, `methodWater` in Listing 4). In this paper, we define a *bounded sea* to be *an island surrounded by context-aware water*.

To automate the definition of bounded seas we introduce a new operator for building tolerant grammars: *the sea operator*. We use the notation `~island~` to create sea from `island`, which can be a terminal or non-terminal. Instead of having to produce complex definitions of sea, a language engineer can use the sea operator which will do the hard work. Listing 5 shows how the grammar of Listing 4 can be defined using the sea operator.

A rule defined with the sea operator (*e.g.*, `~method~`) maintains the composability property of the advanced grammar since by applying the sea operator

we search for the island in a restricted scope. Moreover, such a rule is reusable, robust, and simple to define.

Bounded seas are based on two ideas:

1. *Water never consumes any input from the right context of the bounded sea*, *i.e.*, any input that can appear after the bounded sea. This is very different from the water of "traditional" island grammars, where water is not guaranteed to not consume a part of a valid input (cf. Section 2.2.1). The water of bounded seas is unambiguous, thus improving composability.

2. *Everything is fully automated.* The sea is created using the sea operator `~island~`. Once the sea is placed in the grammar, the grammar is analyzed and appropriate water is created without user interaction. This way the sea can be placed in any grammar. In case the grammar is changed, the water is recomputed automatically. Automatic water computation eases grammar definition, and ensures robustness and reusability of rules.

Bounded seas can be integrated into a parser combinator framework, a highly modular framework for building a parser from other composable parsers [9]. The fact that a bounded sea can be implemented as a parser combinator demonstrates its composability and flexibility.

### *3.2. The Sea Boundary*

Ideally water should never consume any input that can appear after a bounded sea, *i.e.*, it should never consume an input from its right context. We will call the right context the *boundary* of a sea.

The right context of the sea consists of the inputs accepted by parsing expressions that appear after the island. In the case of `A ← ~'a'~ (B / C)`, the right context of `~'a'~` is any input accepted either by `B` or by `C`.

Being aware of the boundary, a tolerant parser can search for methods in a class without the risk that other classes will interfere. Bounded seas would correctly parse the input in Listing 3 because water of a method sea would not be allowed to consume `endclass`, which is a boundary of the `methodSea`.

The island-specific water has to stop in two cases: first, when an island is reached; second, when a boundary is reached. If a boundary is reached before an island is found, the sea fails. The fact that sea can fail implies that sea can be embedded into optional or repetition expressions without ambiguous results. For example, we can define the superclass specification as an optional island:

```
~classDef~ ~superclassSpec~? classBody 'endclass'
```

If `superclassSpec` is not present for the particular class, it will simply fail upon reaching `classBody` instead of searching for `superclassSpec` further and further. The same holds for repetitions.

```
classBody ← ~method~*
```

This rule will consume only methods until it reaches `"endclass"` in the input string, since `endclass` is in the boundary of `~method~`, so methods in another class cannot be inadvertently consumed.

We first define bounded seas generally, and subsequently provide a PEG-specific definition.

**Definition 1** (Bounded Sea)**.** *A* bounded sea *consists of a sequence of three parsing phases:*

1. ***Before-Water:*** *Consume input until an island or the right context appears. Fail the whole sea if we hit the right context. Continue if we hit an island.*
2. ***Island:*** *Consume an island.*
3. ***After-Water:*** *Consume input until the right context is reached.*

### *3.3. The Context Sensitivity of Bounded Seas*

In order to preserve the unambiguity of water in bounded seas, they need to be context-sensitive. A bounded sea recognizes different substrings of an input depending on what surrounds the sea. There are two cases where context-sensitivity emerges:

1. A bounded sea recognizes different input depending on what immediately follows the sea.

2. A bounded sea recognizes different input depending on what immediately precedes the sea.

Let us demonstrate context sensitivity of bounded seas using rules from Listing 6 and two inputs, `"..a..b.."` and `"..a..c.."`. On its own, `A` recognizes any input with `'a'` and `B` recognizes any input with `'b'` (see rows 1-4 in Table 1), because they are not bounded by anything.

```
A  ← ∼'a'∼
B  ← ∼'b'∼

R1 ← A          R2 ← B
R3 ← A 'b'      R4 ← A 'c'
R5 ← A  B
```

Listing 6: Rules for demonstrating context-sensitive behavior.

| | Rule | Input | Result | |
|---|---|---|---|---|
| 1 | `R1 ← A` | `"..a..b.."` | `A` recognizes '..a..b..' | |
| 2 | `R1 ← A` | `"..a..c.."` | `A` recognizes '..a..c..' | |
| 3 | `R2 ← B` | `"..a..b.."` | `B` recognizes '..a..b..' | |
| 4 | `R2 ← B` | `"..a..c.."` | `B` fails | |
| 5 | `R3 ← A 'b'` | `"..a..b.."` | `A` recognizes '..a..' | `'b'` recognizes 'b' |
| 6 | `R3 ← A 'b'` | `"..a..c.."` | `A` recognizes '..a..b..' | `'b'` fails |
| 7 | `R4 ← A 'c'` | `"..a..b.."` | `A` recognizes '..a..b..' | `'c'` fails |
| 8 | `R4 ← A 'c'` | `"..a..c.."` | `A` recognizes '..a..' | `'c'` recognizes 'c' |
| 9 | `R5 ← A B` | `"..a..b.."` | `A` recognizes '..a..' | `B` recognizes 'b..' |
| 10 | `R5 ← A B` | `"..a..c.."` | `A` recognizes '..a..c..' | `B` fails |

Table 1: The seas `A` and `B` recognize different inputs depending on the context.

However, when the two islands are not alone, their boundary can differ, depending on the context. The right context of `A` is `'b'` in `R3`, and the right

context of `A` is `'c'` in `R4`. Therefore `A` consumes different substrings of input depending whether it is called from `R3` or `R4` (see rows 5-8 in Table 1).

A more complex case of context-sensitivity, which we call the *overlapping sea problem*, arises when one sea is immediately followed by another. Consider, for example, rule `R5`, where the sea `A` has as its right context `B`, which is also a sea. Note that the before-water of `B` should consume anything up to its island `'b'` or its own right context, *including the island of its preceding sea* `A`. Now, the before-water of `A` should consume anything up to either its island `'a'` or its right context `B`. But the very search for the right context will now consume the island we are looking for, since `B`'s before-water will consume `'a'`! We must therefore take special care to avoid a "shipwreck" in the case of overlapping seas by disabling the before-water of the second sea. Therefore `B` recognizes `"..a..b.."` when called from `R2` and `"b.."` when called from `R5` (see rows 3 and 9 in Table 1). For the detailed example of the `∼a∼ ∼b∼` sequence, see Appendix B.3.

## 4. Bounded Seas in Parsing Expression Grammars

Starting from the standard definition of PEGs (see Appendix A), we now show how to add the sea operator to PEGs while avoiding the overlapping sea problem. To define the sea operator, we first need the following two abstractions:

1. **The water operator** *consumes uninteresting input.* Water ( `≈` ) is a new PEG prefix operator that takes as its argument an expression that specifies when the water ends. We discuss this in detail in subsection 4.1.

2. **The NEXT function** *approximates the boundary of a sea.* Intuitively, $NEXT(e)$ returns the set of expressions[3] that can appear directly after a particular expression $e$. The details of the NEXT function are given in subsection 4.2.

[3]The NEXT function is modelled after FOLLOW sets from parsing theory, except that instead of returning a set of tokens, it returns a set of parsers.

**Definition 2** (Sea Operator)**.** *Given the definitions of* `≈` *and NEXT, we define the sea operator as follows:* `∼e∼` *is a sequence expression*

```
≈(e / next1 / next2 / ... nextn)
 e
≈(next1 / next2 / ... nextn)
```

*where* $next_i \in NEXT(e)$ *for* $i = 1..n$ *and* $n = |NEXT(e)|$.

That is, the before-water consumes everything up to the island or the boundary, and the after-water consumes everything up to the boundary.

### *4.1. The* Water *Operator*

The purpose of a water expression is to consume uninteresting input. Water consumes input until it encounters the expression specified in its argument (*i.e.*, the *boundary*). We must, however, take care to avoid the overlapping sea problem.

If two seas overlap (one sea is followed by another), the right boundary of the first sea starts with the second sea. Yet it should only start with the island of the second sea as illustrated in subsection 3.3. In order to do so, the second sea has to simply disable its before-water.

We detect overlapping seas as follows: if sea $s_2$ is invoked from the water of another sea $s_1$, it means that the water of $s_1$ is testing for its boundary $s_2$ and thus $s_2$ has to disable its before-water. To distinguish between nested seas (*e.g.*, `∼'x' ∼island∼ 'x'∼`) and overlapping seas (*e.g.*, `∼'x'∼ ∼'y'∼`), we test the position where this sea was invoked. In case of nested seas the positions differ, and in case of overlapping seas they are the same.

**Definition 3** (Extended Semantics of PEGs)**.** *In order to detect overlapping seas and to compute the NEXT set, we extend the original semantics of a PEG* $G = \{N, T, R, e_s\}$ *(see Definition 8 in Appendix A) with a stack of invoked expressions and their positions. For standard PEG operators there is no change except that an explicit stack* $S$ *is maintained. We define a relation* $\Rightarrow$ *from tuples of the form* $(x, S)$ *to the output* $o$*, where* $x \in T^*$ *is an input string to be*

*recognized, $S$ is a stack of tuples $(e, p)$, where $e$ is a parsing expression and $p \geq 0$ is a position, and $o \in T^* \cup \{f\}$ indicates the result of a recognition attempt. The distinguished symbol $f \notin T$ indicates failure. Function $len(x)$ returns the length of an input $x$. Function $(e, p) : S$ denotes a stack with tuple $(e, p)$ on the top and stack $S$ below. $S$ is initialized with the pair $(e_s, 0)$.*

*We define $\Rightarrow$ inductively as follows (without any semantic changes for standard PEG operators):*[4]

$$\textbf{Empty:}\ \frac{x \in T^*}{(x, (\epsilon, p) : S) \Rightarrow \epsilon}$$

$$\textbf{Terminal (success case):}\ \frac{a \in T \qquad x \in T^*}{(ax, (a, p) : S) \Rightarrow a}$$

$$\textbf{Terminal (failure case):}\ \frac{a \neq b \qquad (a, \epsilon, S) \Rightarrow f}{(bx, (a, p) : S) \Rightarrow f}$$

$$\textbf{Nonterminal:}\ \frac{A \leftarrow e \in R \qquad (x, (e, p) : S) \Rightarrow o}{(x, (A, p) : S) \Rightarrow o}$$

$$\textbf{Sequence (success case):}\ \frac{\begin{array}{c}(x_1x_2y, (e_1, p) : S) \Rightarrow x_1 \\ (x_2y, (e_2, p + len(x_1)) : S) \Rightarrow x_2\end{array}}{(x_1x_2y, (e_1e_2, p) : S) \Rightarrow x_1x_2}$$

$$\textbf{Sequence (failure case):}\ \frac{(x, (e_1, p) : S) \Rightarrow f}{(x, (e_1e_2, p) : S) \Rightarrow f}$$

$$\textbf{Sequence (failure case 2):}\ \frac{\begin{array}{c}(xy, (e_1, p) : S) \Rightarrow x \\ (y, (e_2, p + len(x)) : S) \Rightarrow f\end{array}}{(xy, (e_1e_2, p) : S) \Rightarrow f}$$

$$\textbf{Alternation (case 1):}\ \frac{(xy, (e_1, p) : S) \Rightarrow x}{(x, (e_1/e_2, p) : S) \Rightarrow x}$$

$$\textbf{Alternation (case 2):}\ \frac{\begin{array}{c}(x, (e_1, p) : S) \Rightarrow f \\ (x, (e_2, p) : S) \Rightarrow o\end{array}}{(x, (e_1/e_2, p) : S) \Rightarrow o}$$

$$\textbf{Repetitions (repetition case):}\ \frac{\begin{array}{c}(x_1x_2y, (e, p) : S) \Rightarrow x_1 \\ (x_2, (e*, p + len(x_1)) : S) \Rightarrow x_2\end{array}}{(x_1x_2y, (e*, p) : S) \Rightarrow x_1x_2}$$

$$\textbf{Repetitions (termination case):}\ \frac{(x, (e, p) : S) \Rightarrow f}{(x, (e*, p) : S) \Rightarrow \epsilon}$$

[4]Note that in these rules $p$ is implicitly defined as the current position in the input.

$$\textbf{Not predicate (case 1):}\ \frac{(xy, (e, p) : S) \Rightarrow x}{(xy, (!e, p) : S) \Rightarrow f}$$

$$\textbf{Not predicate (case 2):}\ \frac{(xy, (e, p) : S) \Rightarrow f}{(xy, (!e, p) : S) \Rightarrow \epsilon}$$

A detailed example can be found in Appendix B.3.

**Definition 4** (Water Operator)**.** *With the extended semantics of PEGs we can define a prefix* ***water operator*** $\approx$ *. It searches for a boundary and consumes input until it reaches a boundary. If the water starts a boundary of another sea, it stops immediately. Function* $seasOverlap(S, p_1)$ *returns true if there is a pair* $(\approx e, p_2)$ *on a stack* $S$ *where* $p_1 = p_2$ *and* $e$ *is any parsing expression and returns false otherwise.* $x \in T^*$*,* $y \in T^*$*,* $z \in T^*$*.*

$$\textit{Overlapping seas:}\ \frac{seasOverlap(S, p)}{(x, (\approx e, p) : S) = \epsilon}$$

$$\textit{Boundary found:}\ \frac{\begin{array}{c}(yz, (e, p) : S) \Rightarrow y \\ (x'', (e, p + len(x')) : (\approx e, p + len(x')) : S) \Rightarrow f \\ \forall x = x'x''x'''\end{array}}{(xyz, (\approx e, p) : S) = x}$$

In case of *directly nested seas* (*e.g.*, `∼∼island∼∼`) we obtain the same behaviour as with `∼island∼`. The function *seasOverlap* returns true in case a sea is directly invoked from another sea without consuming any input. Applying the rule *Overlapping seas* from Definition 4, water of the inner sea is eliminated and the boundary is the same for the both seas. Therefore `∼∼island∼∼` is equivalent to `∼island∼`.

### *4.2. The NEXT function*

Any input that can appear after the sea forms a boundary of a sea. The NEXT function returns a set of expressions that can appear directly after a particular expression.

Consider the grammar in the example from Listing 7. The `code` rule is defined in such a way that it accepts an arbitrary number of class and structure islands in the beginning (classes and structures can be in any order) and there

is a main method at the end. Intuitively, another class island, a structure island or a main method can appear after a class island.

The NEXT set approximates the boundary. Its expressions recognize prefixes of the boundary and not necessarily the whole boundary. The reason for using NEXT is the limited backtracking ability of PEGs. PEGs are not capable of taking globally correct decisions because they are not able to revert choices that have already been taken.[5].

```
code        ← (~class~/~struct~)* mainMethod
class       ← 'class'  ID classBody
stuct       ← 'struct' ID sbody
mainMethod  ← 'public' 'method' 'main' block

classBody   ← ...
sbody       ← ...
block       ← ...
ID          ← ...
```

Listing 7: Definition of code that consists of classes and structures followed by main method.

For practical reasons, elements of NEXT cannot accept an empty string. For example, an optional expression is not a suitable approximation of a boundary, because it succeeds for any input. Consider a simple expression `~e~ 'a'? 'b'`. The `'a'?` can appear after the *'island'* but `'b'` as well if `'a'` fails. Therefore NEXT has to return `'a'? 'b'`, not just `'a'?`.

We will use *abstract simulation* [3] in order to recognize an expression that accepts an empty string.

**Definition 5** (Abstract Simulation)**.** *We define a relation* $\rightharpoonup$ *consisting of pairs* $(e, o)$*, where* $e$ *is an expression and* $o \in \{0, 1, f\}$*. If* $e \rightharpoonup 0$*, then* $e$ *can succeed on some input while consuming no input. If* $e \rightharpoonup 1$*, then* $e$ *can succeed on some input while consuming at least one terminal. If* $e \rightharpoonup f$*, then* $e$ *may fail on some input. We will use variable* $s$ *to represent a* $\rightharpoonup$ *outcome of either* 0 *or* 1*. We will define the simulation relation* $\rightharpoonup$ *as follows:*

[5]See for example: `http://www.webcitation.org/6YrGmNAi7`

1. $\epsilon \rightharpoonup 0$.

2. $t \rightharpoonup 1$, $t \in T$.

3. $t \rightharpoonup f$, $t \in T$.

4. $A \rightharpoonup o$ *if* $e \rightharpoonup o$ *and* $A \leftarrow e$ *is a rule of the grammar* $G$.

5. $e_1 e_2 \rightharpoonup 0$ *if* $e_1 \rightharpoonup 0$ *and* $e_2 \rightharpoonup 0$.
   $e_1 e_2 \rightharpoonup 1$ *if* $e_1 \rightharpoonup 1$ *and* $e_2 \rightharpoonup s$.
   $e_1 e_2 \rightharpoonup 1$ *if* $e_1 \rightharpoonup s$ *and* $e_2 \rightharpoonup 1$.

6. $e_1 e_2 \rightharpoonup f$ *if* $e_1 \rightharpoonup f$

7. $e_1 e_2 \rightharpoonup f$ *if* $e_1 \rightharpoonup s$ *and* $e_2 \rightharpoonup f$.

8. *(a)* $e_1 / e_2 \rightharpoonup 0$ *if* $e_1 \rightharpoonup 0$
   *(b)* $e_1 / e_2 \rightharpoonup 1$ *if* $e_1 \rightharpoonup 1$

9. $e_1 / e_2 \rightharpoonup o$ *if* $e_1 \rightharpoonup f$ *and* $e_2 \rightharpoonup o$.

10. $e* \rightharpoonup 1$ *if* $e \rightharpoonup 1$

11. $e* \rightharpoonup 0$ *if* $e \rightharpoonup f$

12. $!e \rightharpoonup f$ *if* $e \rightharpoonup s$

13. $!e \rightharpoonup 0$ *if* $e \rightharpoonup f$

Because this relation does not depend on the input string, and there are a finite number of expressions in a grammar, we can compute this relation over any grammar [3]. An example of abstract simulation can be found in Appendix B.1.

**Definition 6** (NEXT)**.** *Let* $S$ *be a stack of* (expression, position) *pairs representing positions and invoked parsing expressions, where* $\overline{\triangle}(S)$ *pops an element from the stack* $S$ *returning a stack* $S'$ *without the top element,* $s_n, s_{n-1}, ..s_2, s_1$ *are expressions on the stack* $S$ *(top of the stack is to the left, bottom to the*

*right),* `$$` *is a special symbol signaling end of input, and* $E_1 \times E_2$ *is a product of two sets of parsing expressions,* $E_1$ *and* $E_2$*, such that* $E_1 \times E_2 = \{e_i e_j | e_i \in E_1, e_j \in E_2\}$*, we define* $NEXT(S)$ *as a set of expressions such that:*

- *if* $s_n = e_1$ *and* $s_{n-1} = e_1 e_2$ *and* $e_2 \not\rightarrow 0$ *then* $NEXT(S) = \{e_2\}$
- *if* $s_n = e_1$ *and* $s_{n-1} = e_1 e_2$ *and* $e_2 \rightharpoonup 0$ *then* $NEXT(S) = \{e_2\} \times NEXT(\overline{\triangle}(S))$
- *if* $s_n = e_1$ *and* $s_{n-1} = e_1 e_2$ *then* $NEXT(S) = \{e_2\}$
- *if* $s_n = e_2$ *and* $s_{n-1} = e_1 e_2$ *then* $NEXT(S) = NEXT(\overline{\triangle}(S))$
- *if* $s_n = e_1$ *or* $s_n = e_2$ *and* $s_{n-1} = e_1 / e_2$ *then* $NEXT(S) = NEXT(\overline{\triangle}(S))$
- *if* $s_n = e$ *and* $s_{n-1} = e*$ *then* $NEXT(S) = e \cup NEXT(\overline{\triangle}(S))$
- *if* $s_n = e$ *and* $s_{n-1} = !e$ *then* $NEXT(S) = \{\}$
- *if* $s_n = e \in N$ *then* $NEXT(S) = NEXT(\overline{\triangle}(S))$
- *if* $n = 0$ *(stack is empty) then* $NEXT(S) = \{$ `$$` $\}$

An example of NEXT computation can be found in Appendix B.2.

## 5. Implementation

As a validation of bounded sea composability and reusability we report on an implementation of bounded seas in the PetitParser framework.[6] The bounded sea extension of PetitParser is part of Moose — a platform for software and data analysis.[7]

[6] `http://scg.unibe.ch/research/IslandParsing/CLSS2015`

[7] `http://moosetechnology.org`

### 5.1. PetitParser Internals

PetitParser [2, 10] is a PEG-based parser combinator [4] framework utilizing scannerless parsing [11] and packrat parsing [12]. Implementations of PetitParser exist for Pharo Smalltalk[8] (the version we extended), Java[9] and Dart.[10]

PetitParser combinators are subclasses of the `PPParser` class, which defines an abstract method `parse:anInput`. If parsing fails, `PPFailure` is returned, otherwise a result is returned. For example, the `PPSequence` combinator is subclass of `PPParser`, having two extra instance variables referring to two parsers that should be in sequence as you can see in Listing 8. The method `parse:anInput` is implemented as shown in Listing 9. The method returns a failure if either of the two parsers fails, and returns both results in an array if they both succeed.

```
PPParser subclass: #PPSequence
    "Sequence of two parsers, p1 and p2"
    instanceVariables: 'p1 p2'.
```

Listing 8: `PPSequence` has two instance variables, `p1` and `p2`.

```
PPSequence>>parse: anInputStream
    | result1 result2 |
    result1 ← p1 parse: anInputStream.
    result1 ifFailure: [ ↑ result1 ].
    result2 ← p2 parse: anInputStream.
    result2 ifFailure: [ ↑ result2 ].

    "return array with both results"
    ↑ { result1 . result2 }
```

Listing 9: Implementation of `PPSequence>>parse:` in PetitParser.

### 5.2. Implementation of BoundedSeas in PetitParser

To support bounded seas, we changed the interface of the `parse: anInput` method to `parse: aPPContext`. `PPContext` is an object that provides access to the stack of invoked expressions. `PPContext` as well implements the

[8] `http://smalltalkhub.com/#!/~Moose/PetitParser`
[9] `https://github.com/petitparser/java-petitparser`
[10] `https://github.com/petitparser/dart-petitparser`

interface of the `InputStream` so that it can be used as `InputStream`. In order to manage the stack of invoked expressions, a parser is dispatched via `PPContext>>invoked:` and a value is returned via `PPContext>>return:` or `PPContext>>fail:`.

`PPBoundedSea` is defined as in Listing 10. Even though a bounded sea consists of a sequence of three parsers, it has only one instance variable `island`, before-water and after-water being created dynamically depending on the state of `PPContext`. The `parse:` method of `PPBoundedSea` is in Listing 11. The three phases of `parse:` correspond to the phases in Definition 1. In order to detect an overlapping sea, there is a check in `parseBeforeWater:` (see Listing 12).

```
PPParsersubclass: #PPBoundedSea
    instanceVariables: 'island'.
```

Listing 10: `PPBoundedSea` has only one instance variable `island`, before and after-water are created dynamically, depending on the state of the `PPContext`.

```
PPBoundedSea>>parse: aPPContext
    | result1 result2 result3 |
    aPPContext invoked: self.
    "Phase One"
    result1 ← self parseBeforeWater: aPPContext.
    result1 ifFailure: [
        ↑ aPPContext fail: 'boundary or island not found'
    ].

    "Phase Two"
    result2 ← island parse: aPPContext
    result2 ifFailure: [
        ↑ aPPContext fail: 'island not found'
    ]

    "Phase Three"
    result3 ← self parseAfterWater: aPPContext.
    result3 ifFailure: [
        ↑ aPPContext fail: 'boundary not found'
    ].

    ↑ aPPContext return: { result1 . result2 . result3 }
```

Listing 11: Implementation of a `parse:` method in `PPBoundedSea`. The three phases corresponds to the phases in the Definition 1.

```
PPBoundedSea>>parseBeforeWater: aPPContext
    | next |
    "Catch Overlapping Seas Problem"
    aPPContext seasOverlap ifTrue: [
        ↑ nil
    ].
    next ← aPPContext next.
    ↑ self goUpTo: island / next.
```

Listing 12: Implementation of a `beforeWater:` method in `PPBoundedSea`.

`PPContext` manages the parsing expression invocation stack, computes the next set and detects the overlapping seas. Thanks to the fact that the method invocation stack can be accessed in the Pharo environment, `PPContext` can reuse the method invocation stack to access the invoked expressions. Because the method invocation stack does not contain the invoked position, `PPContext` manages this separately and only for `PPBoundedSea` parsers (see Listing 13). Overlapping seas can be detected trivially (see Listing 14). The NEXT function implementation follows straightforwardly from recursive Definition 6 (see Listing 15).

```
PPContext>>invoked: parser
  self assert: parser isBoundedSea.
  self invokedPositions push: self position.

PPContext>>return: parser
  self assert: parser isBoundedSea.
  self invokedPositions pop.
```

Listing 13: Implementation of a `invoked:` method and `return:` method in `PPContext`.

```
PPContext>>seasOverlap
  ↑ self invokedPositions top ==
    self invokedPositions secondTop
```

Listing 14: Implementation of a `seasOverlap` method in `PPContext`.

```
PPContext>>next
  | stack |
  stack ← self expressionStackFrom: thisContext.
  ↑ self next: stack into: Set new
```

```
PPContext>>next:stack into: set
  "first sequence case: e1 on top, e1e2 second top,
    e2 isNotNullable then NEXT = {e2}"
  (stack secondTop isSequence and:
  [ stack secondTop first == stack top ] and:
  [ stack secondTop second isNullable not ]) ifTrue: [
    set add: stack secondTop second.
    ↑ set
  ]
  ...
  "repetition case"
  (stack secondTop) isRepetition ifTrue: [
    set add: stack pop.
    ↑ self next: stack into: set
  ]
```

Listing 15: Fragment of a `next` method in `PPContext`.

*5.3. Performance*

In this section we briefly report on the performance of bounded seas. We focus on the time complexity of the three different placements of a sea: standalone seas, repetition of a sea and a nested sea.

We performed measurements on the following parsers and inputs:

1. *Stand-alone sea* `~'a'~` searches for the island `"a"` in an input. An input consists of randomly generated string of dots `.` (representing water) and a single character `"a"` at a random position.

2. *Repetition of a sea* `~'a'~ +` searches for sequences of islands `"a"` in an input. An input consists of a randomly generated string of dots `.` (for water) and island characters `"a"`, *e.g.*, `"..a.....a....a...aa.."`.

3. *Nested sea* `block ← ~'{' block+ / ~ε~ '}'~ +` searches for sequences of nested blocks in an input. An input consists of block starting with `"{"` and ending with `"}"`. A block contains a possibly empty sequence of other blocks, *e.g.*, `"{...{}.{.{..}.}...}"`.

Figure 1 shows that the time complexity is linear compared to the input size for a stand-alone sea and a repetition of a sea. For the nested sea, we mea-

sured an exponential complexity. All of the measured parsers used a memoized version [12, 13] of bounded seas.

Figure 2 reports on equivalents of bounded seas implemented without using the bounded sea operator. The complexity of these parsers is linear. We see that there is room for improvement and this still remains an open issue.

In the case studies we performed (section 7 and section 8), bounded seas showed performance comparable to the non-bounded seas versions. We assume that the typical size of the input and parser complexity used in the case studies is below the threshold where the exponential complexity manifests itself. To support our assumption, Figure 3 compares a bounded sea parser and a non-bounded sea equivalent. Both parsers extract Java methods from Java standard library files (details about extracting Java methods are provided in section 7) with a comparable time performance.

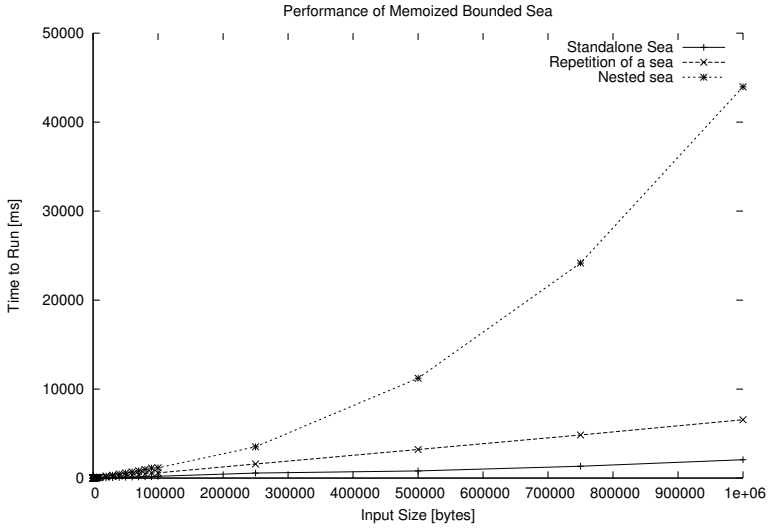


Figure 1: The performance comparison of memoized bounded seas for a stand-alone sea `~'a'~`, a repetition of a sea `~'a'~ +` and for nested sea on randomly generated inputs.

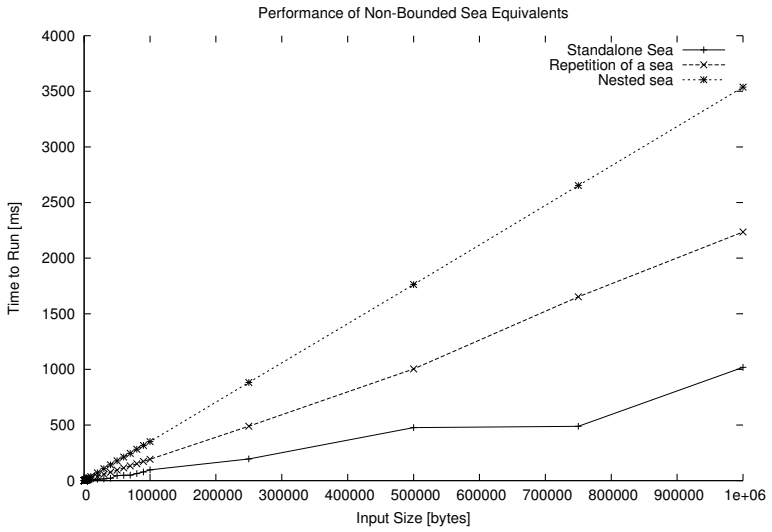


Figure 2: The performance comparison of non-bounded sea equivalents for a stand-alone sea `~'a'~` , a repetition of a seas `~'a'~ +` and for a nested sea on randomly generated inputs.

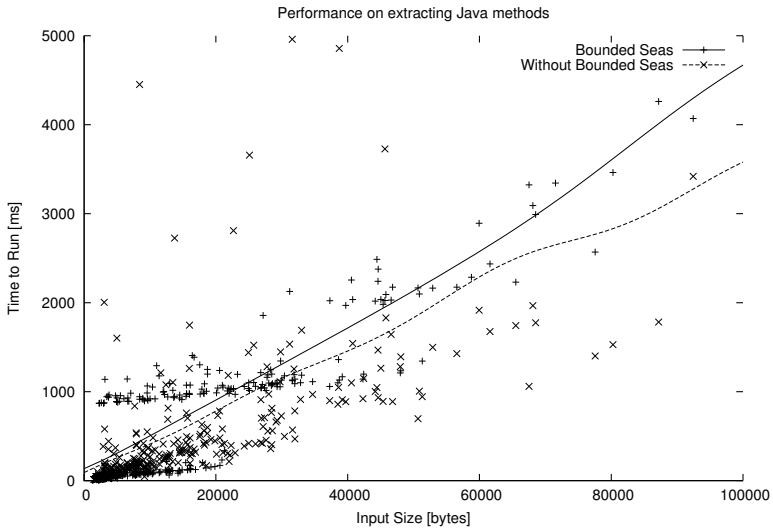


Figure 3: The performance comparison of a bounded sea parser and its non-bounded sea equivalent on extracting Java methods from approximately four hundred files from the Java standard library.

## 6. Discussion

In this section we discuss some implementation decisions of bounded seas as well as an implementation of bounded seas for generalized LL grammars and parser combinator libraries. We also discuss why the NEXT set is needed instead of the simpler FOLLOW sets.

### *6.1. Bounded Seas as Meta-syntactic Sugar*

Bounded seas can be implemented in two ways: as a meta-syntactic sugar or as a parser extension. In this work we take the latter approach. There are several reasons why we did not choose a grammar transformation that transforms a sea expression into a standard PEG expression.

First, it is easier to implement the NEXT function and detect overlapping seas during parsing than detecting overlapping seas statically and transforming the boundary in such a way that the overlapping seas problem cannot arise. Second, PetitParser is a very agile framework where a parser can be updated simply by changing an object reference at any time. Furthermore, the graph of parser combinators corresponds exactly to the grammar, which makes PetitParser easy to understand and debug. Grammar transformation would add an extra level of complexity into the implementation and it would complicate comprehension and debugging.

### *6.2. Generalized LL Parsing*

In this paper we have discussed bounded seas for PEGs. However, the essence of bounded seas is not in the grammar formalism used but in the fact that water is specific for each island and it is computed automatically from a stack of invoked expressions. We argue that bounded islands are useful for Context Free Grammars (CFGs) [14] as well.

The key difference between PEGs and CFGs is that CFGs may return ambiguous results whereas PEGs cannot. Implementing an island grammar as a CFG may lead to ambiguous results even though only one of the results is

desired. The undesired, remaining results are present only because of vaguely-defined water. This is problematic since it is hard to decide which of the results is the correct one.

Bounded seas eliminate ambiguities by adopting a more precise definition of water. Water of a bounded sea never consumes any input that might be valid in a given parsing context. Even though we define a bounded sea with an island `'y'` and we run such a rule on the input `"xyzy"`, the water of the bounded sea consumes only `"x"`, never `"xyz"`, thus avoiding ambiguities.

Generalized LL Parsing [15] can handle any CFG, allows all the choices of CFGs to be explored in parallel, and, in case of ambiguity returns all possible results. Bounded seas can be implemented in a GLL parser because their top-down nature allows for a stack of parsing expressions and they support syntactic predicates used in a boundary.

### *6.3. Integration With Monadic Parser Libraries*

To compute $NEXT(e_1)$ in a sequence $e_1e_2$ we need to know what $e_2$ is. However in some cases, *e.g.*, in monadic parser combinator [4] libraries, $e_2$ could be a closure. For example, when parsing the HTTP header containing a value indicating the length of a content[11], we might read that value and use it to create the parser that reads the content itself (*i.e.*, by length-times reading a character):

```
length >>= \length -> applyNTimes length readChar
```

Now, if we want to use a bounded sea to extract the length, *i.e.*,

```
∼length∼ >>= \length -> applyNTimes length readChar
```

we cannot determine the boundary of `∼length∼`, because it depends on the result of the `length`. As a consequence we can only use bounded seas in a sequence $e_1e_2$ if we can compute $e_2$ before parsing the $e_1$.

[11] `https://en.wikipedia.org/wiki/List_of_HTTP_header_fields`

Bounded seas do not allow for context-sensitive dependencies between an island and its border, but for one exception: when a sea is bounded by another sea, we disable water if another water is already invoked at the same position.

### *6.4. FOLLOW vs. NEXT*

The NEXT function introduces extra complexity into bounded seas, even though it resembles the FOLLOW function from LL parsing theory [16, pp. 235-361]. The key difference between FOLLOW and NEXT is that the former returns only terminals, while the latter returns parsing expressions.

Why is it not sufficient to use the well-known FOLLOW sets instead of the more complicated NEXT function? The reason is that the right context (boundary) of a sea is in general an $LL(k), k \geq 1$ language, and a simple FOLLOW set is not usually sufficient to recognize the boundary.

As an example, consider the grammar from Listing 7. The boundary of `class` is $NEXT($ `class` $) = \{$ `~class~` , `~struct~` , `mainMethod` $\}$. Suppose that instead we take as the boundary of `class` its FOLLOW set, *i.e.*, $FOLLOW($ `class` $) = \{$ `'class'` , `'struct'` , `'public'` $\}$. If there are other elements in the input that start with `'public'` (*e.g.*, `"public int i = 0;"`), they will be indistinguishable from the `mainMethod` and the water of bounded seas would finish in an invalid position.

Bounded seas are supposed to work only with a skeleton of an original grammar with as little information as possible. Therefore, information about other input that can interfere with a boundary (*e.g.*, `"public int i = 0;"`) is not usually available. If bounded seas are are provided with a baseline grammar this would not be problem as the techniques described by Klusener and Lämmel [17] can then be applied.

## 7. Java Parser Case Study

The goal of this case study is to demonstrate the suitability of bounded seas for extracting data from Java sources without any baseline grammar provided.

First we focus on a simpler task without considering nested classes. Because bounded seas target extensibility we subsequently investigate the effort required to extend the parser with nested classes.

We compare four kinds of Java parsers and we measure how well can they extract classes and their methods from a Java source code.[12]

1. **PetitJava** is an open-source Java parser using PetitParser [2] provided by the Moose analysis platform community [18]. We used version 159.[13]

2. **Naïve Island Parser** is an island parser with water defined simply as the negation of the island we are searching for. The sea rules in this parser can be reused, because they do not consider their surroundings and they are grammar-independent. The sea rules are defined in a simple form: consume input until an island is found, then consume an island.

3. **Advanced Island Parser** is a more complex version of the naïve island parser. The water is more complicated to prevent the most frequent failures of island parsers. The sea rules in this parser are hard-wired to the grammar and cannot be reused. The sea rules are customized for a particular islands.

4. **Island Parser with Bounded Seas** is an island parser written using bounded seas. The sea rules were defined using the sea operator.

The PetitJava parser parses Java 6 code. All the island parsers (island, advanced and bounded) are very similar, with approximately 20 rules per each. PetitJava itself contains over 200 rules. The island parsers were designed to extract classes and the methods that belong to them. None of the parsers was optimized to provide a better performance.

We compare the three island parsers (almost identical in a structure) written by the first author. It is very likely that the advanced island parser can be

[12]The case study and instructions can be found at the following prepared web-page: `http://scg.unibe.ch/research/IslandParsing/CLSS2015`.

[13]`http://smalltalkhub.com/#!/~Moose/PetitJava/`

modified to achieve better precision and better performance, but at the cost of considerable engineering work. We demonstrate that naïve water rules do not work and that the advanced version of water is needed. We further show that with bounded seas we can obtain high precision and performance without needing to define an advanced island parser. Finally, we show that extending an island parser is a highly demanding task, unless bounded seas are used.

*Test Data.* For our case study we randomly selected 50 files ($N$) containing 50 classes from the JDK 6 library. These 50 classes contain 81 nested classes and a total of 1380 methods $M$. We extract the reference data using the VerveineJ[14] parser.

Each parser returns a set $m$ of fully-qualified method names[15], some of which are true positives $m_{tp}$. If a parser fails, an error is returned and the set of all errors is $e$. Failure is treated as though no classes or methods were found. We measure precision $P = |m_{tp}|/|m|$), recall $R = |m_{tp}|/|M|$, error rate $err = |e|/N$ and time per file $t = t_{total}/N$.

### *7.1. Without Nested Classes*

First of all, we evaluate our parsers on extracting method names without considering the nested classes and their methods. We can easily skip the nested classes by defining properly paired blocks starting with `'{'` and ending with `'}'` and ignoring everything inside.

*Results.* As we see in Table 2, PetitJava parser provides perfect precision, but recall is poor because of the high error rate.[16] On the other hand, the error rate of all island parsers (island, advanced and bounded) is very low,[17] but precision and recall are not perfect, even though they are relatively good. Amongst the imprecise parsers, the Bounded parser provides the best precision and recall.

[14] `https://gforge.inria.fr/projects/verveinej`

[15] `http://docs.oracle.com/javase/specs/jls/se7/html/jls-6.html#jls-6.7`

[16] The PetitJava failures are due to bugs in the grammar specification.

[17] Failures of the imprecise parsers are due to parsing timeout (set to 10 seconds).

| Parser | Precision | Recall | Time [ms] | Error Rate |
|---|---|---|---|---|
| PetitJava | 1.00 | 0.71 | 308 | 0.28 |
| Island | 0.87 | 0.90 | 1225 | 0.04 |
| Advanced | 0.92 | 0.90 | 1336 | 0.04 |
| Bounded | 0.96 | 1.00 | 941 | 0.00 |

Table 2: Precision, recall error rate and time of the four tested parsers without considering nested classes.

### *7.2. With Nested Classes*

In this step, we extend our island parsers to include nested classes and their methods. We do this by making a single change, where we extend the `classBody` rule from this[18]:

```
classBody ← '{' method island * '}'
```

to this:

```
classBody ← '{' (method / class) island * '}'
```

| Parser | Precision | Recall | Time [ms] | Error Rate |
|---|---|---|---|---|
| PetitJava | 1.00 | 0.67 | 299 | 0.28 |
| Island | 0.87 | 0.54 | 934 | 0.12 |
| Advanced | 0.94 | 0.32 | 1734 | 0.34 |
| Advanced′ | 0.91 | 0.68 | 847 | 0.03 |
| Bounded | 0.97 | 0.99 | 627 | 0.00 |

Table 3: Precision, recall, time and error rate including nested classes.

*Results.* As we see in Table 3 the PetitJava parser performs as in the previous case. Yet the imprecise parsers (Island, Advanced) start to struggle. Their error rate has increased and recall has dropped dramatically. The errors were mostly

[18] `island` here creates either an island, an advanced island or a bounded sea depending on the parser we use.

due either to parsing timeouts (when parsing took more than ten seconds per file) or various parsing errors. On the other hand, the Bounded parser maintains high precision and recall, zero error rate, and improves time per file slightly.

In Table 3 we also measured the Advanced′ parser, which made use of refined rules for water to take into account the grammar changes.[19] This improved recall, parsing time and the error rate. We would, however, need to invest even more effort to reach the quality of the Bounded parser.

## 8. Ruby Parser Case Study

The standard approach to recognize the structure of the input is to track all language elements that affect structure, as we did in the Java case study where we defined a rule for blocks. As it turns out, almost anything can affect the structure of a Ruby program. For this reason, we turned to indentation as it turns out to be a good proxy for structure [19]. In this case study we focus on using bounded seas to extract the structure of a Ruby program by exploiting indentation.[20]

### *8.1. The Dangling End Problem*

Ruby poses interesting parsing challenges even for imprecise parsers. The biggest problem we faced is the *dangling end problem*: Normally a control structure like an if statement terminates with `end`. However there is also an *if modifier*, as in `return error if check?`, which does not require an `end`.

Such modifiers pose problems for parsing. There exist numerous such modifiers in Ruby[21], which resemble conditional blocks, but have a different syntax.

[19] We investigated the reasons for failures and added an extra boundary to `classBody`.

[20] The case study and instructions can be found at the prepared web-page:
`http://scg.unibe.ch/research/IslandParsing/CLSS2015`.

[21] `http://docs.huihoo.com/ruby/ruby-man-1.4/syntax.html#if-mod`
`http://docs.huihoo.com/ruby/ruby-man-1.4/syntax.html#unless-mod`
`http://docs.huihoo.com/ruby/ruby-man-1.4/syntax.html#while-mod`
`http://docs.huihoo.com/ruby/ruby-man-1.4/syntax.html#until-mod`

From the perspective of an imprecise parser, it is hard to distinguish between a modifier, loops and conditional blocks.

Ruby structures (such as classes, methods, blocks) end with the `'end'` keyword (see Listing 16). To capture the structure of Ruby code, we need to define rules for these structural elements, including conditional blocks and others, such as loops, do blocks, and brace pairs.

```
class Shape
  def draw
    if (x > 0)
      do_something()
    end
  end
end
```

Listing 16: Example of a Ruby code.

Ruby modifiers are not paired with any `'end'` as we can see in Listing 17. If we incorrectly pair `'end'`, we change the structure of a program. Unfortunately, it is hard to recognize when `'if'` belongs to a modifier and when to a conditional block, unless we specify a complete grammar to recognize all the constructs that it could possibly modify.

```
class Shape
  def draw
    return error if check?
    if (x > 0)
      do_something
    end
  end
end
```

Listing 17: Example of Ruby code where `'if'` is not paired with any `'end'`.

### *8.2. Indentation*

It is known that indentation is a good proxy for structure in programming languages [19]. We can exploit this fact to define a context-sensitive parser that uses both indentation and bounded seas to recognize modifiers. From the perspective of indentation, modifiers look like loops or conditional blocks with a single line scope.

Because PetitParser produces scannerless parsers [11] and it doesn't use any preprocessing (*i.e.*, tokenizing), an indentation-sensitive parser is context-sensitive since the question whether code is indented or not depends on the results of previously invoked parsers.

Inspired by Landin's offside rule [20], indentation in PetitParser uses a stack of indentation levels and adds extra layout-oriented parsing expressions (*e.g.*, align, inOffside). These expressions consult the stack and the current indentation level to verify that the input complies with the given layout criteria [21].

Although we have seen earlier that bounded seas are incompatible with monadic parser libraries where a boundary may depend on what has been parsed earlier, indentation parsing is a special case that does not interfere with bounded seas. As we shall see, the use of indentation-sensitive parsers simplifies the implementation of the parsers and even improves the overall performance.

We define a context-sensitive grammar that recognizes modules, classes, methods and class methods in Ruby code by utilizing indentation and bounded seas. The scope of a class or method extends as far as code appears to the right of the class or method declaration (*i.e.*, in the onside position). The `class` definition is in Listing 18.

```
class ← setOffsideLine, 'class' identifier
         ∼(class / method)∼ onside *
        unsetOffsideLine
```

Listing 18: Indentation Sensitive definition of a Ruby class.

### *8.3. Parsing Results*

In this section we report on the complexity, performance, precision and recall of three parsers: a classical *island parser* (46 grammar rules, 9K characters), a *bounded parser* that does not utilize indentation (41 rules, 8.5K characters), and an *indent bounded parser* that utilizes indentation (27 rules, 4K characters).

The island parser and the bounded parser are almost identical. The sea parser uses bounded seas, while the island parser uses manually defined islands and water. From the number of methods, we can see that indentation simplifies

```
method    ← 'def' name arguments primary* 'end'
primary   ← (!(comment / keyword / modifier / ...)
             #any)*
            (method / class)
arguments ← ...
```

Listing 20: Method definition in Island Grammar.

the implementation. The bounded parser and the island parser must implement additional rules to recognize the dangling end.

The bounded parser shows its flexibility here. For example, the method definition in the bounded grammar does not require `arguments` (Listing 19) contrary to the method definition in the island grammar (Listing 20).

```
methodDef ← 'def' name primary* 'end'
primary   ← ~method / class~
```

Listing 19: Method definition in the Bounded Grammar.

To measure precision and recall, we used *jruby-parser*[22] as a reference parser. We compared the structure (modules, classes, methods and class methods) of Ruby code as detected by jruby-parser with the structure detected by our parsers. We describe the structure as a set of methods where each method is prepended with a path consisting of other methods, classes and modules depending on the location of the method in a code, similar to Java's fully qualified names. For example:

```
<module>graphics.<class>Shape.<method>draw
```

refers to a method `draw` defined in the class `Shape`. The `Shape` belongs to the `graphics` module. On the other hand:

```
<class>Shape.<class>Renderer.<class-method>Instance
```

refers to the class-side method `Instance` of the `Shape`'s inner class `Renderer`.

[22] `https://github.com/jruby/jruby-parser`

*Test Data.* We performed our study on a sample of $N = 100$ files of six popular projects on Github: Rails[23], Discourse[24], Diaspora[25], Cucumber[26], Valgrant[27] and Typhoeus.[28] The sampled files contain a total of 520 methods.

Parsers return a set of fully qualified methods $m$, where some of them are true positives $m_{tp}$. If a parser fails, an error is returned. The set of all errors is $e$. We measure precision $P = |m_{tp}|/|m|)$, recall $R = |m_{tp}|/|M|$, error rate $err = |e|/N$ and time per file $t = t_{total}/N$. Failure is treated as though no classes or methods are found.

| **Parser** | **Precision** | **Recall** | **Time [ms]** | **Error Rate** |
|---|---|---|---|---|
| Island Parser | 1.00 | 0.96 | 495 | 0.03 |
| Bounded Parser | 0.97 | 0.96 | 283 | 0.01 |
| Indent Bounded Parser | 0.99 | 0.99 | 203 | 0.00 |

Table 4: Precision, recall error rate and time of compared parsers.

Table 4 shows precision and recall are rather high in all of the cases. The island parser has perfect precision, but recall is not perfect due to some failures. The bounded parser has worse precision, because it did not fail for one of the inputs, but misplaced the methods into the wrong module. The indent bounded parser can parse any of the files with very high precision and recall. It misplaced only one[29] of all the methods.

As we have seen, the island parser contains 46 rules, the bounded parser 41, and the indent parser 27. This shows that both bounded seas and indentation help to reduce the complexity of the Ruby grammar. Bounded seas perform better than traditional islands. The indentation parser is even better than the

[23] `https://github.com/rails/rails`

[24] `https://github.com/discourse/discourse`

[25] `https://github.com/diaspora/diaspora`

[26] `https://github.com/cucumber/cucumber`

[27] `https://github.com/mitchellh/vagrant`

[28] `https://github.com/typhoeus/typhoeus`

[29] If a method declaration with a modifier follows an inner class defined on a single line, the method with the modifier is incorrectly assigned to the inner class.

bounded parser, because fewer rules are needed to determine the boundaries.

## 9. Related Work

*Agile Parsing.* Agile parsing [7] is a recent paradigm for source analysis and reverse engineering tools. In agile parsing the effective grammar used by a particular tool is a combination of two parts: the standard base grammar for the input language, and a set of explicit grammar overrides that modify the parse to support the task at hand. There are several agile parsing idioms: i. *rule abstraction* (grammar rules can be parametrized); ii. *grammar specialization* (grammar rules can be specialized based on the semantic needs); iii. *grammar categorization* (to deal with context-free ambiguities); iv. *union of grammars* (to unify multiple grammars); v. *markup* (to match and mark chunks of interest); vi. *semi-parsing* (to define islands and lakes); and vii. *data structure grammars* (separate grammars that hold auxiliary data structures).

The semi-parsing idiom [7] uses the *not* predicate to prevent water from consuming islands. This approach is the same as that taken by bounded seas. Contrary to the semi-parsing idiom, bounded seas are able to infer the predicates on their own. The agile parsing idioms are based on a transformation of a well-defined baseline grammar, whereas bounded seas do not expect such a well-defined grammar and must infer the predicates only from the available *skeleton*.

*Island Grammars.* Island grammars were proposed by Moonen [1] as a method of semi-parsing to deal with irregularities in the artifacts that are typical for the reverse engineering domain. Island grammars make use of a special syntactic rule called *water* that can accept any input. Water is annotated with a special keyword `avoid` that will ensure that water will be accepted only if there is no other rule that can be applied.

Contrary to Moonen, we propose boundaries (based on the NEXT function) that limit the scope in which water can be applied. Because each island has a

different boundary, our solution does not use the single water rule; instead our water is tailored to each particular island.

*Non-Greedy Rules.* Non-greedy operators are well-known from regular expressions introduced in Perl.[30] `??`, `*?`, and `+?` are non-greedy versions of `?`, `*` and `+`, which match as little of a string as possible while preserving the overall match. The backtracking algorithm admits a simple implementation of non-greedy operators: try the shorter match before the longer one. For example, in a standard backtracking implementation, `e?` first tries using `e` and then tries not using it; `e??` uses the other order.[31]

Non-greedy operators are also available in ANTLR as parser operators. A non-greedy parser matches the shortest sequence of tokens that preserves a successful parse for a valid input sentence. Contrary to regular expressions, a non-greedy parser never makes a decision that will ultimately cause valid input to fail later on during the parse. The central idea is to match the shortest sequence of tokens that preserves a successful parse for a valid input sentence.[32]

Bounded seas are distinct from non-greedy rules in two ways. First, bounded seas do not require globally correct decisions, since they are not available in traditional PEGs. Though PEGs can backtrack while choosing between alternatives, once the choice is made it cannot be changed, thus making a globally correct decision impossible. In order to realize non-greedy repetitions, PEGs feature predicates, which have to be specified by an engineer (as illustrated in section 2). Bounded seas remove the burden of predicates from a language engineer by computing the NEXT set automatically.

Second, bounded seas target transparent composability. A language engineer can treat a bounded sea like any other PEG rule without bothering about its implementation. For example, the following grammar can be eas-

[30] `http://perldoc.perl.org/perlre.html`

[31] `https://swtch.com/~rsc/regexp/regexp1.html`

[32] `https://theantlrguy.atlassian.net/wiki/display/ANTLR4/Wildcard+Operator+and+Nongreedy+Subrules`

ily modified by changing the `body` to `body ← sea*`, `body ← sea?` or `body ← sea? sea?`.

```
start ← ('begin' body 'end')*
body  ← sea
sea   ← ~sea~
```

If we define `sea` using lazy repetition `*?`, the normal `sea` can be defined as:

```
start ← ('begin' body 'end')*
body  ← sea
sea   ← .*? 'body' .*?
```

the optional version as:

```
start ← ('begin' body)*
body  ← sea
sea   ← .*? ('body'|'end')
```

the repetition version as:

```
start ← ('begin' body 'end')*
body  ← sea
sea   ← ('body'|.)*?
```

and the sequence of two optional seas as:

```
start ← ('begin' body)*
body  ← sea1
sea1  ← .*?('body1' sea2   | 'body2' 'end'  | 'end')
sea2  ← .*?('body2' 'end' | 'end')
```

*Noise Skipping Parsing.* GLR* is a noise-skipping parsing algorithm for context-free grammars able to parse any input sentence by ignoring unrecognizable parts of the sentence [22]. The parser nondeterministically skips some words in a sentence and returns the parse with fewest skipped words. The parser is a modification of Generalized LR (Tomita) parsing algorithm [23].

The GLR* application domain is parsing of spontaneous speech. Contrary to bounded seas, GLR* itself decides what is noise (water in our case) and where it is. In the case of bounded seas the positions of the noise (water) are explicitly defined.

*Fuzzy Parsing.* The term fuzzy parser was coined for Sniff [24], a commercial C++ IDE that uses a hand-made top-down parser. Sniff can process incomplete programs or programs with errors by focusing on symbol declarations (classes, members, functions, variables) and ignoring function bodies. In linguistics or natural language processing [25], the notion of fuzzy parsing corresponds to an algorithm that recognizes fuzzy languages.

The semi-formal definition of a fuzzy parser was introduced by Koppler [26]. Fuzzy parsers recognize only parts of a language by means of an unstructured set of rules. Compared with whole-language parsers, a fuzzy parser remains idle until its scanner encounters an anchor in the input or reaches the end of the input. Thereafter the parser behaves like a normal parser. In the fuzzy parsing framework, islands can occur in any order, always start with a terminal and everything between them is ignored; in the bounded seas paradigm, the islands are constrained by the context-free structure.

*Skeleton Grammars.* Skeleton grammars [17] address the issue of false positives and false negatives when performing tolerant parsing by inferring a tolerant (skeleton) grammar from a precise baseline grammar.

Our approach tackles the same problem as skeleton grammars: improving the precision of island grammars. They both maintain the composability property and both can be automated. Skeleton grammars use the standard first and sets known from standard parsing theory [16, pp. 235-361] for synchronization with the baseline grammar.

Bounded seas do not require a precise baseline grammar and they have to find point of synchronization based only on the the main grammar itself. Therefore the main grammar has to contain all the relevant information (*e.g.*, when extracting classes and methods with bounded seas block definitions are essential to place methods properly). Because the main grammar of bounded seas is typically far from complete, bounded seas use the NEXT set (instead of first and follow) to reach the required precision. If bounded seas are provided with the baseline grammar, the boundaries can be computed from the baseline.

*Bridge Parsing.* Bridge parsing is a novel, lightweight recovery algorithm that complements existing recovery techniques [27]. Bridge parsing extends an island grammar with the notion of bridges and reefs. Islands denote tokens that open or close scopes. Reefs are attributed tokens and they add information (*e.g.*, indentation) to nearby islands. Islands and reefs are created in a tokenizing phase. Bridges connect matching opening and closing islands in a bridge-building phase. The corresponding islands are searched with the help of reefs (*e.g.*, indentation can be used to find matching brackets). If some islands are not connected (*e.g.*, if the opening or closing scope island is missing), the bridge repair phase tries to repair them with the help of information from reefs.

The focus of bounded seas is on data extraction rather than on error recovery and bounded seas are missing advanced error-recovery techniques available in the bridge parsing. Bounded seas are meant to be used on valid inputs without errors. If an erroneous chunk appears, bounded seas skip such a chunk until a valid chunk is found. To our best knowledge, techniques used in bridge parsing are complementary to bounded seas and might help improve precision of bounded seas on erroneous inputs.

*Permissive Grammars.* The main idea of permissive grammars [28, 29] is to derive a permissive grammar from a standard grammar. Such a permissive grammar accepts programs with minor errors (missing brackets, *etc.*). A permissive grammar is also a normal grammar and can be tweaked by the language engineer. Using a specialized version of the GLR algorithm, both syntactically correct and incorrect programs can be efficiently parsed using these grammars [28].

Contrary to bounded seas, which target the area of rapid data extraction, permissive grammars are supposed to help IDE developers with interactive parsing and error recovery as the user is writing a program. Similarly to bounded seas, permissive grammars extend the concept of island grammars and use water for error recovery. Even though bounded seas can be used to skip over noise in an input, bounded sea handle missing or misspelled input simply by ignoring the whole erroneous chunk until a valid chunk is found. Permissive grammars

try to find the best way to fix an erroneous chunk (and not only skip over it).

## 10. Conclusion

In this paper we have presented bounded seas — composable, reusable, robust and easy to use islands. Contrary to the traditional approach of island parsing, bounded seas compute the scope within which water can consume the input. We have extended the semantics of PEGs to implement useful and practical bounded seas. Boundaries are computed by a NEXT function, inspired by the follow function from standard parsing theory. The automation of the process that creates the bounded sea ensures that bounded seas are easy to use and are not error-prone. Bounded seas as presented in this work are context-sensitive.

As a validation of the composability and reusability of bounded seas, we have presented an implementation of bounded seas as a parser combinator in the PetitParser framework. Furthermore we have presented two case studies applying bounded sea parsers to extracting method names from Java and Ruby code, and we have compared these parsers to conventional parsers based on a precise grammar and based on island grammars. We show that bounded seas provide both good precision and performance.

*Acknowledgments*

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assessment" (SNSF project No. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015).

We also thank the anonymous referees for their invaluable comments.

## References

[1] L. Moonen, Generating robust parsers using island grammars, in: E. Burd, P. Aiken, R. Koschke (Eds.), Proceedings Eighth Working Conference on Reverse Engineering (WCRE 2001), IEEE Computer Society, 2001, pp. 13–22. doi:10.1109/WCRE.2001.957806.

URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.13.1027

[2] L. Renggli, S. Ducasse, T. Gîrba, O. Nierstrasz, Practical dynamic grammars for dynamic languages, in: 4th Workshop on Dynamic Languages and Applications (DYLA 2010), Malaga, Spain, 2010, pp. 1–4.
URL http://scg.unibe.ch/archive/papers/Reng10cDynamicGrammars.pdf

[3] B. Ford, Parsing expression grammars: a recognition-based syntactic foundation, in: POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, New York, NY, USA, 2004, pp. 111–122. doi:10.1145/964001.964011.
URL http://pdos.csail.mit.edu/~baford/packrat/popl04/peg-popl04.pdf

[4] G. Hutton, E. Meijer, Monadic parser combinators, Tech. Rep. NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham (1996).
URL citeseer.ist.psu.edu/hutton96monadic.htmlhttp://eprints.nottingham.ac.uk/237/1/monparsing.pdf

[5] P. Klint, E. Visser, Using filters for the disambiguation of context-free grammars, in: Proc. ASMICS Workshop on Parsing Theory, 1994, pp. 1–20.

[6] M. van den Brand, J. Scheerder, J. J. Vinju, E. Visser, Disambiguation filters for scannerless generalized LR parsers, in: N. Horspool (Ed.), Compiler Construction (CC'02), Vol. 2304 of Lecture Notes in Computer Science, Springer-Verlag, Grenoble, France, 2002, pp. 143–158.
URL http://www.cs.uu.nl/people/visser/ftp/BSVV02.pdf

[7] T. R. Dean, J. R. Cordy, A. J. Malton, K. A. Schneider, Agile parsing in TXL, Autom. Softw. Eng. 10 (4) (2003) 311–336.
URL http://research.cs.queensu.ca/~cordy/Papers/JASE_AP.pdf

[8] V. Zaytsev, Formal foundations for semi-parsing, in: Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on, 2014, pp. 313–317. doi:10.1109/CSMR-WCRE.2014.6747184.
URL http://grammarware.net/text/2014/semiparsing.pdf

[9] R. Frost, J. Launchbury, Constructing natural language interpreters in a lazy functional language, Comput. J. 32 (2) (1989) 108–121. doi:10.1093/comjnl/32.2.108.
URL https://courses.cit.cornell.edu/ling4424/frost-launchbury.pdf

[10] J. Kurs, G. Larcheveque, L. Renggli, A. Bergel, D. Cassou, S. Ducasse, J. Laval, PetitParser: Building modular parsers, in: Deep Into Pharo, Square Bracket Associates, 2013, p. 36.
URL http://scg.unibe.ch/archive/papers/Kurs13a-PetitParser.pdf

[11] E. Visser, Scannerless generalized-LR parsing, Tech. Rep. P9707, Programming Research Group, University of Amsterdam (Jul. 1997).
URL http://www.cs.uu.nl/people/visser/ftp/P9707.ps.gz

[12] B. Ford, Packrat parsing: simple, powerful, lazy, linear time, functional pearl, in: ICFP 02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming, Vol. 37/9, ACM, New York, NY, USA, 2002, pp. 36–47. doi:10.1145/583852.581483.
URL http://pdos.csail.mit.edu/~baford/packrat/icfp02/packrat-icfp02.pdf

[13] B. Ford, Packrat parsing: a practical linear-time algorithm with backtracking, Master's thesis, Massachusetts Institute of Technology (2002).
URL http://pdos.csail.mit.edu/~baford/packrat/thesis/http://pdos.csail.mit.edu/~baford/packrat/thesis/thesis.pdf

[14] N. Chomsky, Three models for the description of language, IRE Transactions on Information Theory 2 (1956) 113–124, http://www.chomsky.info/articles/195609--.pdf.

[15] E. Scott, A. Johnstone, GLL parsing, Electron. Notes Theor. Comput. Sci. 253 (7) (2010) 177–189. doi:10.1016/j.entcs.2010.08.041.
URL http://dx.doi.org/10.1016/j.entcs.2010.08.041

[16] D. Grune, C. J. Jacobs, Parsing Techniques — A Practical Guide, Springer, 2008.
URL http://www.cs.vu.nl/~dick/PT2Ed.html

[17] S. Klusener, R. Lämmel, Deriving tolerant grammars from a base-line grammar, in: Proceedings of the International Conference on Software Maintenance (ICSM 2003), IEEE Computer Society, 2003, pp. 179–188. doi:10.1109/ICSM.2003.1235420.

[18] O. Nierstrasz, S. Ducasse, T. Gîrba, The story of Moose: an agile reengineering environment, in: Proceedings of the European Software Engineering Conference (ESEC/FSE'05), ACM Press, New York, NY, USA, 2005, pp. 1–10, invited paper. doi:10.1145/1095430.1081707.
URL http://scg.unibe.ch/archive/papers/Nier05cStoryOfMoose.pdf

[19] A. Hindle, M. W. Godfrey, R. C. Holt, Reading beside the lines: Indentation as a proxy for complexity metrics, in: ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension, IEEE Computer Society, Washington, DC, USA, 2008, pp. 133–142. doi:10.1109/ICPC.2008.13.
URL http://swag.uwaterloo.ca/~ahindle/pubs/hindle08icpc.pdf

[20] P. Landin, The next 700 programming languages, Communications of the ACM 9 (3) (1966) 157–166. doi:10.1145/365230.365257.
URL http://www.cs.utah.edu/~eeide/compilers/old/papers/p157-landin.pdf

[21] A. S. Givi, Layout sensitive parsing in the PetitParser framework, Bachelor's thesis, University of Bern (Oct. 2013).
URL http://scg.unibe.ch/archive/projects/Sade13a.pdf

[22] A. Lavie, M. Tomita, GLR* — an efficient noise-skipping parsing algorithm for context free grammars, in: In Proceedings of the Third International Workshop on Parsing Technologies, 1993, pp. 123–134.

[23] M. Tomita, Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems, Kluwer Academic Publishers, Norwell, MA, USA, 1985.

[24] W. R. Bischofberger, Sniff: A pragmatic approach to a C++ programming environment, in: C++ Conference, 1992, pp. 67–82.
URL http://citeseer.nj.nec.com/bischofberger92sniff.html

[25] P. Asveld, A fuzzy approach to erroneous inputs in context-free language recognition, in: Proceedings of the Fourth International Workshop on Parsing Technologies IWPT'95, Institute of Formal and Applied Linguistics, Charles University, Prague, Czech Republic, 1995, pp. 14–25.
URL http://doc.utwente.nl/64694/

[26] R. Koppler, A systematic approach to fuzzy parsing, Software: Practice and Experience 27 (6) (1997) 637–649. doi:10.1002/(SICI)1097-024X(199706)27:6<637::AID-SPE99>3.0.CO;2-3.
URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.4.3198&rep=rep1&type=pdf

[27] E. Nilsson-Nyman, T. Ekman, G. Hedin, Practical scope recovery using bridge parsing, in: D. Gašević, R. Lämmel, E. Van Wyk (Eds.), Software Language Engineering, Vol. 5452 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 95–113. doi:10.1007/978-3-642-00434-6_7.
URL http://dx.doi.org/10.1007/978-3-642-00434-6_7

[28] L. C. L. Kats, M. de Jonge, E. Nilsson-Nyman, E. Visser, Providing rapid feedback in generated modular language environments. Adding error recovery to scannerless generalized-LR parsing, in: G. T. Leavens (Ed.), Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2009), ACM SIGPLAN Notices, ACM Press, New York, NY, USA, 2009.

[29] M. de Jonge, L. C. L. Kats, E. Soderberg, E. Visser, Natural and flexible error recovery for generated modular language environments, ACM Transactions on Programming Languages and Systems 34 (4), article No. 15, 50 pages. doi:10.1145/2400676.2400678.

## Appendix A. Parsing Expression Grammars

PEGs were first introduced by Ford [3] and the formalism is closely related to top-down parsing. PEGs are syntactically similar to CFGs [14], but they have different semantics. The main semantic difference is that the choice operator in PEG is ordered — it selects the first successful match — while the choice operator in CFG is ambiguous. PEGs are composed using the operators in Table A.5.

**Definition 7** (PEG Definition)**.** *We use the standard definition as suggested by Ford [3]. A* parsing expression grammar *(PEG) is a 4-tuple* $G = \{N, T, R, e_s\}$*, where* $N$ *is a set of nonterminals,* $T$ *is a set of terminals,* $R$ *is a set of rules,* $e_s$ *is a start expression.* $N \cap T = \emptyset$*. Each* $r \in R$ *is a pair* $(A, e)$*, which we write* $A \leftarrow e$*, where* $A \in N$*,* $e$ *is a parsing expression. Parsing expressions are defined inductively. If* $e$*,* $e_1$ *and* $e_2$ *are parsing expressions, then so is:*

- $\epsilon$*, the empty string*
- $a$*, any terminal where* $a \in T$
- $A$*, any nonterminal where* $A \in N$
- $e_1 e_2$*, a sequence*

| Operator | Description |
|---|---|
| ′ ′ | Literal string |
| [ ] | Character class |
| · | Any character |
| $(e)$ | Grouping |
| $e?$ | Optional |
| $e*$ | Zero-or-more repetitions of e |
| $e+$ | One-or-more repetitions of e |
| $\&e$ | And-predicate, does not consume input |
| $!e$ | Not-predicate, does not consume input |
| $e_1\ e_2$ | Sequence |
| $e_1$ / $e_2$ | Prioritized choice |

Table A.5: Operators for constructing parsing expressions

- $e_1/e_2$, *a prioritized choice*
- $e*$, *zero or more repetitions*
- $!e$ *a not-predicate*

*The following operators are syntactic sugar:*

- ***Any Character:*** $\cdot$ *is character class containing all letters*
- ***Character class:*** $[a_1, a_2, ...a_n]$ *character class is* $a_1/a_2/../a_n$
- ***Optional expression:*** $e?$ *is* $e_d/\epsilon$, *where* $e_d$ *is desugaring of* $e$
- ***One-or-more repetitions:*** $e+$ *is* $e_d e_d*$, *where* $e_d$ *is desugaring of* $e$
- ***And-predicate:*** $\&e$ *is* $!(!e_d)$, *where* $e_d$ *is desugaring of* $e$

*We will use text in quotation marks to refer to terminals* e.g., `'a'`, `'b'`, `'class'`. *We will use identifiers* `A`, `B`, `C`, `class` *or* `method` *to refer to nonterminals. We will use* `e` *or indexed* `e`: $e_1$, $e_2$, ... *to refer to parsing expressions.*

**Definition 8** (PEG Semantics)**.** *To formalize the semantics of a grammar* $G = \{N, T, R, e_s\}$*, we define a relation* $\Rightarrow$ *from pairs of the form* $(e, x)$ *to the output* $o$*, where* $e$ *is a parsing expression,* $x \in T^*$ *is an input string to be recognized and* $o \in T^* \cup \{f\}$ *indicates the result of a recognition attempt. The distinguished symbol* $f \notin T$ *indicates failure.*

$$\textbf{\textit{Empty:}} \frac{x \in T^*}{(\epsilon, x) \Rightarrow \epsilon}$$

$$\textbf{\textit{Terminal (success case):}} \frac{a \in T, x \in T^*}{(a, ax) \Rightarrow a}$$

$$\textbf{\textit{Terminal (failure case):}} \frac{a \neq b, \quad (a, \epsilon) \Rightarrow f}{(a, bx) \Rightarrow f}$$

$$\textbf{\textit{Nonterminal:}} \frac{A \leftarrow e \in R \quad (e, x) \Rightarrow o}{(A, x) \Rightarrow o}$$

$$\textbf{\textit{Sequence (success case):}} \frac{\begin{array}{c}(e_1, x_1x_2y) \Rightarrow x_1 \\ (e_2, x_2y) \Rightarrow x_2\end{array}}{(e_1e_2, x_1x_2y) \Rightarrow x_1x_2}$$

$$\textbf{\textit{Sequence (failure case 1):}} \frac{(e_1, x) \Rightarrow f}{(e_1e_2, x) \Rightarrow f}$$

$$\textbf{\textit{Sequence (failure case 2):}} \frac{(e_1, x_1y) \Rightarrow x_1 \quad (e_2, y) \Rightarrow f}{(e_1e_2, x_1y) \Rightarrow f}$$

$$\textbf{\textit{Alternation (case 1):}} \frac{(e_1, xy) \Rightarrow x}{(e_1/e_2, x) \Rightarrow x}$$

$$\textbf{\textit{Alternation (case 2):}} \frac{(e_1, x) \Rightarrow f \quad (e_2, x) \Rightarrow o}{(e_1/e_2, x) \Rightarrow o}$$

$$\textbf{\textit{Repetitions (repetition case):}} \frac{\begin{array}{c}(e, x_1x_2y) \Rightarrow x_1 \\ (e*, x_2) \Rightarrow x_2\end{array}}{(e*, x_1x_2y) \Rightarrow x_1x_2}$$

$$\textbf{\textit{Repetitions (termination case):}} \frac{(e, x) \Rightarrow f}{(e*, x) \Rightarrow \epsilon}$$

$$\textbf{\textit{Not predicate (case 1):}} \frac{(e, xy) \Rightarrow x}{(!e, xy) \Rightarrow f}$$

$$\textbf{\textit{Not predicate (case 2):}} \frac{(e, xy) \Rightarrow f}{(!e, xy) \Rightarrow \epsilon}$$

## Appendix B. Examples

### *Appendix B.1. Example of Abstract Simulation*

Let us compute the abstract simulation (see Definition 5) for the following grammar:

```
S   ← E1 E2
E1  ← 'a' / ϵ
E2  ← 'b' 'c'
```

Because of the recursive nature of the definition, we will compute $\rightharpoonup$ for terminals first and we will infer the $\rightharpoonup$ for more complex expressions once we have computed $\rightharpoonup$ for the simpler ones:

- `'a'` $\rightharpoonup$ 1 (rule 2), same for `'b'` and `'c'`
- `'a'` $\rightharpoonup f$ (rule 3), same for `'b'` and `'c'`
- `ϵ` $\rightharpoonup$ 0 (rule 10)
- `E1` $\rightharpoonup$ 0 (rule 9)
- `E2` $\rightharpoonup$ 1 (rule 5)
- `E2` $\rightharpoonup f$ (rule 6)
- `S` $\rightharpoonup$ 1 (rule 5)
- `S` $\rightharpoonup f$ (rule 7)

### *Appendix B.2. Example of NEXT computation*

Let us compute $NEXT$ of the `method` island defined in the island grammar in Listing 5. Let us suppose we have already parsed `"class Foo"` in the input `"class Foo endclass"`. The stack now looks as shown below in Figure B.4.

```
4 | ( ≈ (method/NEXT( ∼method∼ )) method  ≈ (NEXT( ∼method∼ )) ,9) |
3 |                          ( ∼method∼ ,9)                       |
2 |                          ( methodSea ,9)                      |
1 |                          ( methodSea* ,9)                     |
0 |              ( 'class' id methodSea* 'endclass' ,0)           |
```

Figure B.4: State of a stack after parsing `"class Foo"` in the input `"class Foo endclass"`

To parse `∼method∼` we need to compute $NEXT($ `∼method∼` $)$. We do this in the following steps.[33]

1. Initialize: $NEXT($ `methodSea` $) = \{\}, n = 2$
2. Check stack:

   $s_n = s_2=$ `methodSea` and

   $s_{n-1} = s_1 = e*$, where $e =$ `methodSea`
3. Apply rule for $e*$: $NEXT($ `methodSea` $) = \{$ `methodSea` $\} \cup NEXT($ `methodSea` $*)$

   (a) Call: $NEXT($ `methodSea*` $)$

   (b) Initialize: $NEXT($ `methodSea*` $) = \{\}$, n = 1

   (c) Check stack:

   $s_n = s_1 =$ `methodSea*` and

   $s_{n-1} = s_0 = e_1e_2e_3e_4$, $e_3 =$ `methodSea*` , $e_4 =$ `'endclass'`

   (d) Apply the rule for sequence, where $e_4 \not\vdash 0$:

   $NEXT($ `methodSea*` $) = \{$ `'endclass'` $\}$

   (e) Return: $NEXT($ `methodSea*` $) = \{$ `'endclass'` $\}$
4. Return: $NEXT($ `methodSea` $) = \{$ `methodSea 'endclass'` $\}$

*Appendix B.3. PEG Example*

Let us go through the grammar $S \leftarrow$ `∼a∼ ∼b∼` using `"..a..b.."` as an input. As we see in Figure B.5, the stack is initalized with $(S, 0)$ and the whole result is `"..a..b.."`, because it is a result of nonterminal expansion $S \leftarrow$ `∼a∼ ∼b∼` . The sequence on the top is straightforward, as `∼a∼` consumes `"..a.."` and `∼b∼` consumes `"b.."`, and the result is then `"..a..b.."` (see Figure B.6).

In order to get result of `∼a∼` invoked in position 0, we first follow Definition 2 (see Figure B.7). It is a sequence of three parsers (generalization from the sequence of two to the sequence of three is straightforward). In Figure B.8 we

[33]To simplify, we start from stack position 2, because $NEXT($ `∼method∼` $)$ (stack position 3) is trivially $NEXT($ `methodSea` $)$ (stack position 2).

**Nonterminal:**

1. `S ← ∼a∼ ∼b∼` $\in R$

2. $\dfrac{\texttt{..a..b..} \qquad \begin{array}{l}1\\0\end{array}\left|\begin{array}{c}(\ \texttt{∼a∼ ∼b∼}\ ,0)\\(\ \texttt{S}\ ,0)\end{array}\right| \qquad \Rightarrow \quad o = \texttt{..a..b..}}{\texttt{..a..b..} \qquad {\scriptstyle 0}\left|(\ \texttt{S}\ ,0)\right| \qquad \Rightarrow \quad o = \texttt{..a..b..}}$

Figure B.5: The Inference rule for Nonterminal

**Sequence I (success case):**

1. $\texttt{..a..b..} \qquad \begin{array}{l}2\\1\\0\end{array}\left|\begin{array}{c}(\ \texttt{∼a∼}\ ,0)\\(\ \texttt{∼a∼ ∼b∼}\ ,0)\\(\ \texttt{S}\ ,0)\end{array}\right| \qquad \Rightarrow \quad x_1 = \texttt{..a..}$

2. $\texttt{b..} \qquad \begin{array}{l}2\\1\\0\end{array}\left|\begin{array}{c}(\ \texttt{∼b∼}\ ,5)\\(\ \texttt{∼a∼ ∼b∼}\ ,0)\\(\ \texttt{S}\ ,0)\end{array}\right| \qquad \Rightarrow \quad x_2 = \texttt{b..}$

$\dfrac{\texttt{..a..b..} \qquad \begin{array}{l}1\\0\end{array}\left|\begin{array}{c}(\ \texttt{∼a∼ ∼b∼}\ ,0)\\(\ \texttt{S}\ ,0)\end{array}\right| \qquad \Rightarrow \quad x_1x_2 = \texttt{..a..b..}}{}$

Figure B.6: The Inference rule for Sequence

see that before-water consumes `".."`, the island itself consumes the desired `"a"` and another `".."` is consumed by after-water.

**Rewrite according to the Definition 2:**

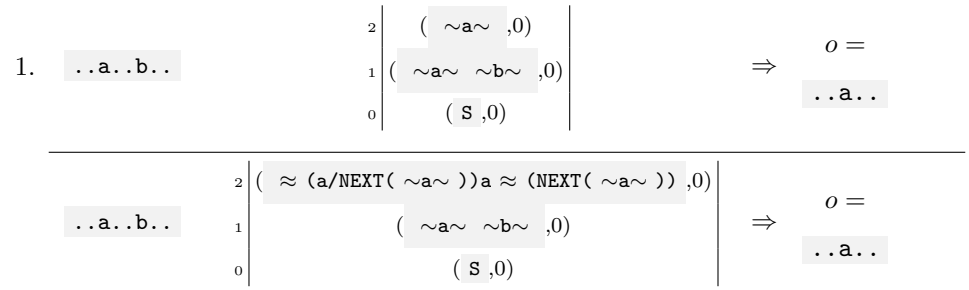


Figure B.7: Rewrite Rule according to the Definition 2

**Sea Sequence I (success case):**

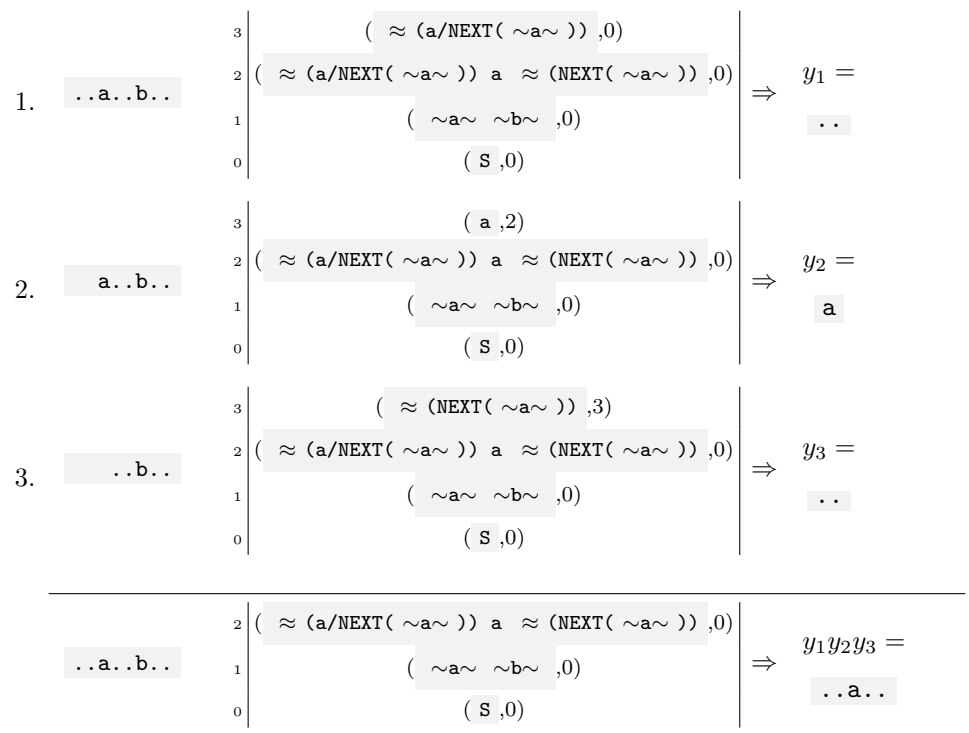


Figure B.8: The Inference rule for Sequence

Let us investigate what happens in before-water of ∼a∼. First of all, we need to determine the NEXT( ∼a∼ ). In this case it is ∼b∼ (see Appendix B.2 for more complex example). Once we know the boundary, before-water tries

to find the island `a` or its boundary `∼b∼` at positions 0 and 1 until it finds the island at the position 2 (see Figure B.9). We return a substring of all the positions for which we failed, *i.e.*, `".."`

**Water (boundary found):**

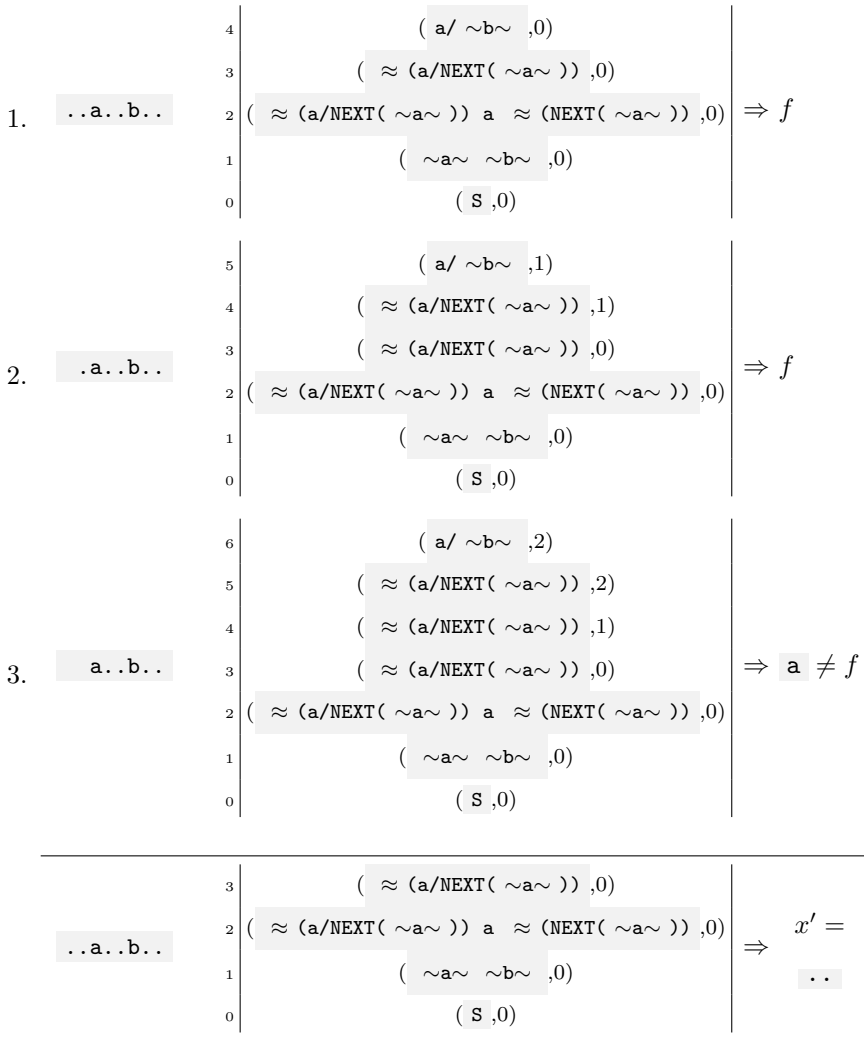


Figure B.9: The Inference rule for Water

*Overlapping Seas.* The interesting question is, why does `∼b∼` fail in position 0? We already explained the problem with overlapping seas in subsection 3.3,

and now we show the computation formally. First of all, we rewrite the sea on top of the stack according to Definition 2. The new sequence on top of the stack fails because before-water returns $\epsilon$ and there is no `b` at position 0 (see Figure B.10) .

**Sea Sequence II (failure case):**

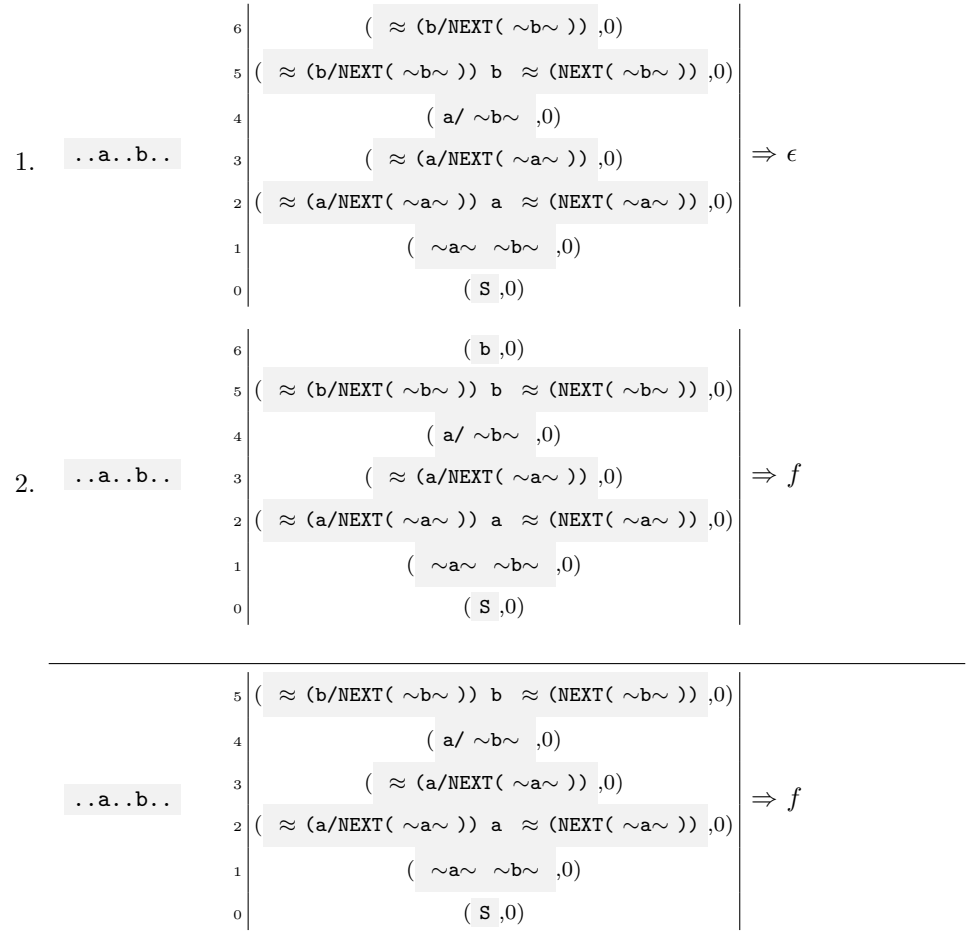


Figure B.10: The Inference rule for Sequence (failure case)

The before-water of `∼b∼` returns $\epsilon$, because of the overlapping seas case. It analyzes the stack and notices the before-water of `∼a∼` invoked on the position 0 (using the `seasOverlap` function) and returns $\epsilon$ (see Figure B.11).

If there is no case of overlapping seas in the grammar, the before-water of

**Water (Overlapping Seas Case):**

$$1.\ \text{seasOverlap}\left(\begin{array}{l|c} 6 & (\ \approx \texttt{(b/NEXT(}\sim\texttt{b}\sim\texttt{))}\ ,0) \\ 5 & (\ \approx \texttt{(b/NEXT(}\sim\texttt{b}\sim\texttt{))}\ \texttt{a}\ \approx \texttt{(NEXT(}\sim\texttt{b}\sim\texttt{))}\ ,0) \\ 4 & (\ \texttt{a/}\sim\texttt{b}\sim\ ,0) \\ 3 & (\ \approx \texttt{(a/NEXT(}\sim\texttt{a}\sim\texttt{))}\ ,0) \\ 2 & (\ \approx \texttt{(a/NEXT(}\sim\texttt{a}\sim\texttt{))}\ \texttt{a}\ \approx \texttt{(NEXT(}\sim\texttt{a}\sim\texttt{))}\ ,0) \\ 1 & (\ \sim\texttt{a}\sim\ \sim\texttt{b}\sim\ ,0) \\ 0 & (\ \texttt{S}\ ,0) \end{array}\right) = true$$

$$\texttt{..a..b..}\quad \left|\begin{array}{l|c} 6 & (\ \approx \texttt{(b/NEXT(}\sim\texttt{b}\sim\texttt{))}\ ,0) \\ 5 & (\ \approx \texttt{(b/NEXT(}\sim\texttt{b}\sim\texttt{))}\ \texttt{a}\ \approx \texttt{(NEXT(}\sim\texttt{b}\sim\texttt{))}\ ,0) \\ 4 & (\ \texttt{a/}\sim\texttt{b}\sim\ ,0) \\ 3 & (\ \approx \texttt{(a/NEXT(}\sim\texttt{a}\sim\texttt{))}\ ,0) \\ 2 & (\ \approx \texttt{(a/NEXT(}\sim\texttt{a}\sim\texttt{))}\ \texttt{a}\ \approx \texttt{(NEXT(}\sim\texttt{a}\sim\texttt{))}\ ,0) \\ 1 & (\ \sim\texttt{a}\sim\ \sim\texttt{b}\sim\ ,0) \\ 0 & (\ \texttt{S}\ ,0) \end{array}\right| \Rightarrow \epsilon$$

Figure B.11: The Inference rule for Overlapping Seas

$\sim$b$\sim$ consumes "..a.." contrary to the correct parse $\epsilon$ (see Figure B.11). This means that the before-water of $\sim$a$\sim$ (see Figure B.9) would be $x' = \epsilon$. This would then fail the whole $\sim$a$\sim$ and consequently the whole $\sim$a$\sim$ $\sim$b$\sim$.